卒業論文

LZ 圧縮回路の設計とハード/ソフト最適分割の検討

氏名　　：中山和也

学籍番号：2260050097-3

指導教員：山崎　勝弘　教授

提出日　：2009 年 2 月 19 日

立命館大学　理工学部電子情報デザイン学科

内容梗概

本論文では、ハードウェア記述言語 Verilog-HDL を用いたアプリケーションの高速化について、また、ハード/ソフト協調設計で課題となっている、ハード/ソフト最適分割について検討した結果について述べる。対象アプリケーションとして、LZ 法を参考にしたデータ圧縮プログラムを用いた。

まず LZ77 のアルゴリズムを理解し、モジュール分割案の検討を行なった。それらをもとに、各モジュールのソフトウェアをコーディングし、ソフトウェアの関数に対応したハードウェア記述を行った。その後、分割されたモジュールを用い、処理部分の性能の測定を行った。

アプリケーションの高速化については、LZ 法のデータ圧縮プログラムをハードウェア化することで、ソフトウェアの約４０倍の速度向上が得られた。

ハード/ソフト最適分割の検討では、ソフトウェアの処理を行った際の負荷をモジュール毎に計測する。ハードウェアについても遅延時間、回路規模等を測定し、それらの結果を考慮したハード/ソフトの分割パターンを決定し、各パターンにおける検討と考察を行った。今回の研究では、LZ 圧縮の処理を、４つのモジュールに分割し、それぞれソフトウェアおよびハードウェアモジュールとして実現し、実験を行った。

目次

図目次

表目次

1. はじめに

半導体技術の進歩に伴い、集積回路は超大規模化、高速化が進んでいる。また、LSI が搭載される応用製品群も多機能化、低価格化、TTM（Time To Market）が進んでおり、これらの電子機器に搭載される集積回路に対する、大規模化、高集積化、設計期間の短縮などへの要求は高まる一方である。

一方、次代の半導体産業を支える柱として、マイクロプロセッサと機器向けの電子回路とを一つの半導体チップ上に搭載した「システム LSI」に対する期待が高まっている。システム LSI は、通常、プロセッサをコンポーネントとして含む。また、そのプロセッサ上で動作するソフトウェアは、ハードウェアと協調して動作するため、ハードウェアとソフトウェアを同時に協調させて設計する必要がある。また、デジタルカメラや携帯電話等の組込みシステムに搭載されているシステム LSI では、ハードウェアとソフトウェアの高度な連携が必要である。

これを実現するための設計手法として、ハード・ソフト協調設計が注目されている。その背景として，次の二つが挙げられる。一つは，半導体製造技術の進歩により，「これまで複数の LSI で構成されていたシステムが一つのチップ上に実現できるようになった」ことである。いわゆる，SoC（system on a chip）が登場した。SoC は回路規模が大きく，かつ複雑な構成になっていることが多い。このため，設計と検証の期間が延びがちである。設計の後段で不具合が見つかると，非常に大きな問題になる。設計の早い段階で，大きな不具合がないことを確認することが重要になっている。

もう一つの理由は，「ハードウェア部とソフトウェア部の関係が密接になった」ことである。例えば，SoC では，ハードウェア部とソフトウェア部が密に連携して，ひとまとまりのデータ処理を行うことが多い。両部を別々に開発したり検証することは，現実的ではない。

これらの二つの背景に加えて，電子機器の開発サイクルが短くなっていることもある。従来に比べて、短期間で確実な開発が求められている。それに応えるのが、ハード・ソフト協調設計である。

本研究では、データ圧縮の一つである LZ 法を対象として、アルゴリズムのハードウェアの記述を作成し、FPGA とソフト・マクロ CPU を用いたシステム上で CPU 処理の負荷の測定を行う。また、有効な分割パターンを検討する。

LZ77 は、1977 年にジェイコブ・ジヴ（Jacob Ziv）とエイブラハム・レンペル（Abraham Lempel）によって開発されたアルゴリズムである。データを先頭から順番に符号化していくスライド辞書方式のデータ圧縮アルゴリズム方式である。LZ77 の改良版は、圧縮ツールの LHA や GZIP などに用いられている。符号化の原理は、現在注目している位置から始まる記号列が、それ以前に出現していたかを探す。 もし出現していたならば、記号列をその出現位置と長さのポインタに置き換える。 記号列を探す範囲をスライド窓と呼び、これを辞書として使用するので、辞書式圧縮法と呼ばれる。この方式の特徴として、同じ内容のデータ列が継続して出現する場合に高い圧縮率を得ることができる。

もともとの LZ77 では、記号列を（一致位置,一致長,次の不一致記号）という 3 つの値に置き換えるが，さまざまな亜種が存在する。 中でも LZSS は、単純で性能もよく、いろいろな応用に使用されている。今回は研究用であるため、複雑な部分を省き、最も原始的な LZ77 アルゴリズムを使用する。

本研究では、LZ 圧縮のアルゴリズムを参考に、C による LZ 圧縮アプリケーションの作成を行った。その後、ハードソフト協調設計に用いることを考慮し、アプリケーションを大まかな処理ごとに 3 つの関数に分割する。次に、作成した関数に対応したハードウェアモジュールを作成する。これらの関数とモジュールを組み合わせ、FPGA 上に構築したシステム上で性能や規模などを計測し、比較することで lz 圧縮を用いた圧縮アルゴリズムにおけるハード/ソフト最適分割の検討を行う。

2. LZ 圧縮

2.1 概要

LZ77 は、1977 年にジェイコブ・ジヴ（Jacob Ziv）とエイブラハム・レンペル（Abraham Lempel）によって開発されたデータ圧縮アルゴリズムの一つである。データを先頭から順番に符号化していく方式であり、スライド辞書方式のデータ圧縮アルゴリズム方式である。このスライド辞書方式は、改良が加えられ以降のデータ圧縮方式にも使用されており、LZ 法には、いくつかの種類がある。その種類によってさらに名称が変わるが、その違いは符号化の方法だけで、処理の内容については全て同じである。

2.2 LZ 圧縮アルゴリズム

LZ77 のアルゴリズムは、以前に出現した文字列を検索し、一致した部分に対して圧縮を行う。例えば、以下の①のようなデータ列の場合、6 文字以降の"ABCD"は以前に出現していた文字列"ABCD"と一致する。一致していた文字列の先頭の位置を記憶しておき、その数値と文字数から、「何文字前から、何文字が一致」と表すことで、②のように[5,4]と置換することができる。さらに続く"ABCDEABCDABCDEABCD"は、③の様に[9,18]と置換することができる。

ＡＢＣＤＥＡＢＣＤＡＢＣＤＥＡＢＣＤＡＢＣＤＥＡＢＣＤ ①

ＡＢＣＤＥ[5.4]　ＡＢＣＤＥＡＢＣＤＡＢＣＤＥＡＢＣＤ ②

ＡＢＣＤＥ[5.4]　[9.18] ③

以前に出現した文字を全て検索すると、データ量が増えるにつれて処理時間も増加してしまうといった恐れがある。そのため、以前に出現した文字列の中から、最も近い固定長の部分データを使って比較処理を行っている。この固定長の部分データは Sliding Window と呼ばれ、処理が進むごとにシフトしていく。これがスライド辞書の処理である。具体的な処理の流れは以下のようになる。[5]

1. Sliding Window の末尾のポインタを、データ列先頭の一つ前にする。
2. データ列先頭から始まるデータ列と一致する部分データ列を Sliding Window から探し、その中で最も長いデータ列を抽出する。
3. もしデータ列が見つからなければ、データそのものを出力して、Sliding Window の終端を一つ後ろにシフトする。
4. データ列が見つかった場合、その位置と長さを出力してから、一致したデータ列の長さ分だけ Sliding Window の終端ををシフトする。

表 1：計算例

	Sliding Window	入力データ列	出力データ列
1.	**********	ABCDEABCDABCDEABCDABCDEABCD	
2.	**********A	BCDEABCDABCDEABCDABCDEABCD	A
3.	**********AB	CDEABCDABCDEABCDABCDEABCD	AB
4.	**********ABC	DEABCDABCDEABCDABCDEABCD	ABC
5.	**********ABCD	EABCDABCDEABCDABCDEABCD	ABCD
6.	**********ABCDE	ABCDABCDEABCDABCDEABCD	ABCDE
7.	*******ABCDEABCD	ABCDEABCDABCDEABCD	ABCDE[5,4]
8.	ABCDEABCDABCDEABCD		ABCDE[5,4][9,18]

上記の例、表 1 では、6 番目の処理において、データ列先頭から 5 つ前にある"ABCDE"の 4 つが一致しているので、[5,4]が出力されている。また、次の 7 番目の処理では、データ列先頭から 9 つ前にある"ABCDEABCDABCDEABCD"の 18 個が一致しているので、[9,18]が出力されている。Sliding Window からはみ出しても、比較処理はそのまま続けている。

逆に、圧縮したデータを展開する場合、以下のように処理を行う。

1. データそのものならば、そのまま出力する。
2. 圧縮されたデータならば、1 番目のパラメータで前に出力されたデータ列内の出力する位置を決めて、2 番目のパラメータであるデータ数分だけ順次コピーしていく。

2.3 他の圧縮アルゴリズムとの比較

圧縮のアルゴリズムは多数存在するが、それぞれの性能を比較すると以下のような特徴をもつ。

(1) 圧縮率

圧縮率については、以下のような順で良い結果が得られる。

LZ78 ≒ LZ77 ＞ 算術符号 ＞ ハフマン符号 ＞ 連長符号

圧縮率については、通常のテキストファイルに処理を行った場合、LZ78 符号と LZ77 符号が４５～50%と、ほぼ半分以下に圧縮することができる。算術符号が 60～70%、ハフマン符号が算術符号よりも数%大きくなり、連長符号は 80～90%程度である。 プログラムなどのバイナリファイルを対象とした場合は、テキストファイルよりも 10～20%ほど大きくなる。

連長符号はテキストファイルやプログラムファイルでは圧縮率が悪く、時には元のファイルよりも大きくなってしまうことさえあるが、同じコードが連続する画像ファイルでは比較的圧縮率が良いので、主に画像データの圧縮に用いられる。

(2) 圧縮速度

圧縮速度については、以下のような順の処理速度となる。

連長符号 ＞ ハフマン符号 ＞ 算術符号 ＞ LZ78 符号 ＞ LZ77 符号

LZ 圧縮は処理が多いため、符号化に時間がかかると言える。そのため、ハードウェアによる高速化のメリットが大きい。

(3)展開速度

連長符号 ＞ ハフマン符号 ＞ LZ77 符号 ＞ 算術符号 ＞ LZ78 符号

圧縮アルゴリズムは圧縮速度よりも展開速度の方が速く、特に LZ77 符号は圧縮速度に比べて展開速度が非常に速いという特徴がある。 それに対して、LZ78 符号は圧縮速度と展開速度がほぼ等しい。

(4)総合処理速度

圧縮と展開を合計した総合処理速度は、以下のような順になる。

連長符号 ＞＞ ハフマン符号 ＞ 算術符号 ＞ LZ78 符号 ＞ LZ77 符号

このような結果となるが、各手法の速度の差はそれほど大きくなく、実質上はほとんど問題にならない程度である。 ハードウェアの進歩により、圧縮／展開速度は日々向上されている。

以上、圧縮率と圧縮／展開速度を総合すると、現在 LZ78 符号が一番多用されている理由が納得できる。

尚、現在普及している圧縮プログラムで LZ77 符号と呼ばれているもののほとんどは、LZSS 符号になっている。

2.4　C 言語による実現

LZ77 アルゴリズムに基づいて、C 言語によるソフトウェア実装を行った。実行環境は Windows 上の Cygwin 上で動作させ、コンパイラは gcc を用いた。

設計したソフトウェアの実行時間、クロック数等の計測を行った。ソフトウェア実行環境は Core2Duo 3.0Ghz、Dram2.0GB、WindowsXP、Cygwin 上である。入力として、45kb の HTML 文章を与えた。計測を行った結果を表 2 に示す。また、アプリケーション実行時に CPU にかかる負荷の割合を図 1 に示す。Loop 関数の処理時間が 6 割を占めており、Out 関数についてはほぼ処理時間が 0 に近い。処理結果として、45kb のソースファイルを 27kb まで圧縮できた。

表 2：ソフトウェアの実行時間

関数名	Mem	Loop	Out	total
処理内容	テキストファイルの格納	圧縮処理	圧縮信号の作成と出力	-
実行時間(秒)	0.031	0.047	0.001	0.079
実行クロックサイクル数(MClock)	93	141	3	235

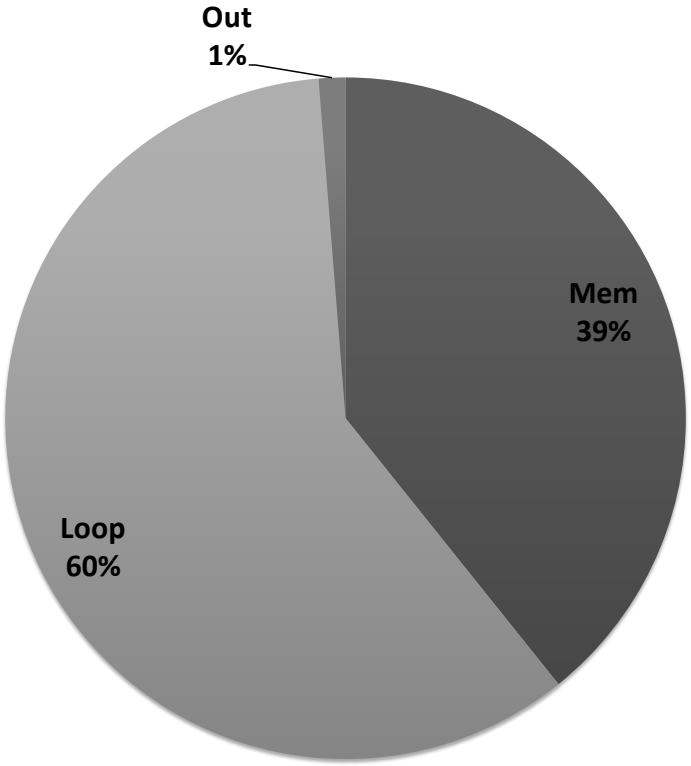


図 1：アプリケーションの関数ごとの CPU 負荷割合

3. LZ 圧縮のハードウェア化

3.1 アルゴリズムの状態遷移図

LZ 圧縮アルゴリズムをハードウェアモジュールとして実現した。処理の流れとしては、図 2 のように行われる。テキストファイルを読み込んだ後、配列を順にシークしていき、以前に同じ単語があったかを判定する。同じ文字が一致している限り、状態は EQUAL のままである。条件が満たされない場合、出力の状態となり、圧縮情報として出力するかどうかの判定をして、出力が行われた後に Equal に戻り、処理が終了するまで Equal と Output で遷移を繰り返す。

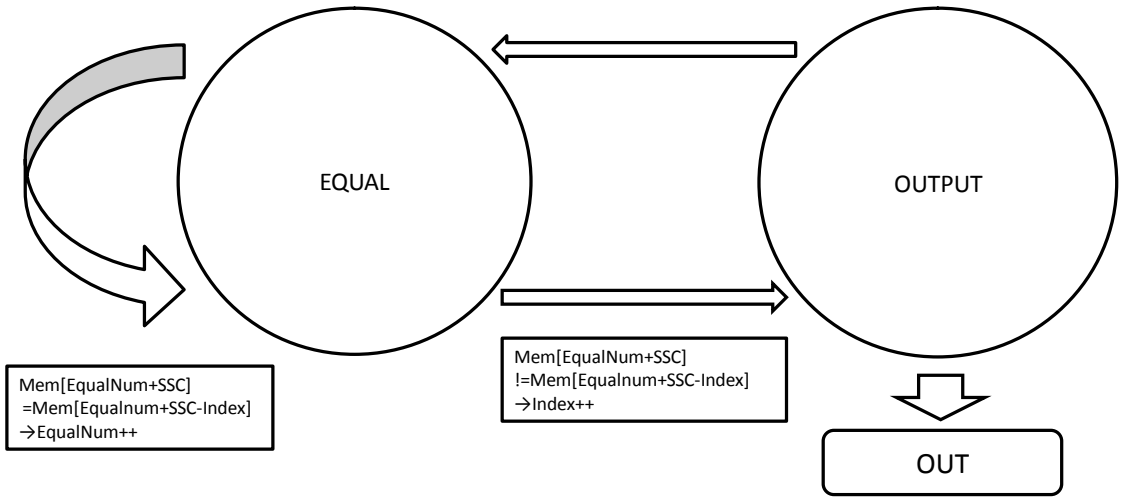


図 2：状態遷移図

3.2　モジュール

分割方法を図 3 に、各信号の詳細を表 3 に示す。モジュールを大きく分けて 4 つのモジュールに分割した。各モジュールについて、データの流れと共に説明する。

top モジュールに、図 3 に示すような Mem,Loop,Out モジュールを接続する。Mem モジュールで、HTML 等のテキストファイルから、ASCII 文字の読み込みを行う。読み込まれた文字を Mem 内のレジスタに保存しておく。また、ソースの文字数を SRCSIZE として計算し、各モジュールに出力する。Loop からの EqualNum,Index をもとに、現在参照している文字を ram1 として出力し、一致しているかどうか検索する文字を ram2 として Loop に出力する。Out からの SSC をもとに、ram3 の出力を行う。ソースファイルの読み込みが完了した時点で、Comp_Start 信号が立ち上がり、圧縮処理が開始される。Loop モジュールでは、圧縮処理の要となる、スライド辞書処理を行う。Out モジュールでは、Loop モジュールで計算した値を利用して、元のデータを圧縮情報に置き換えたデータを出力する。

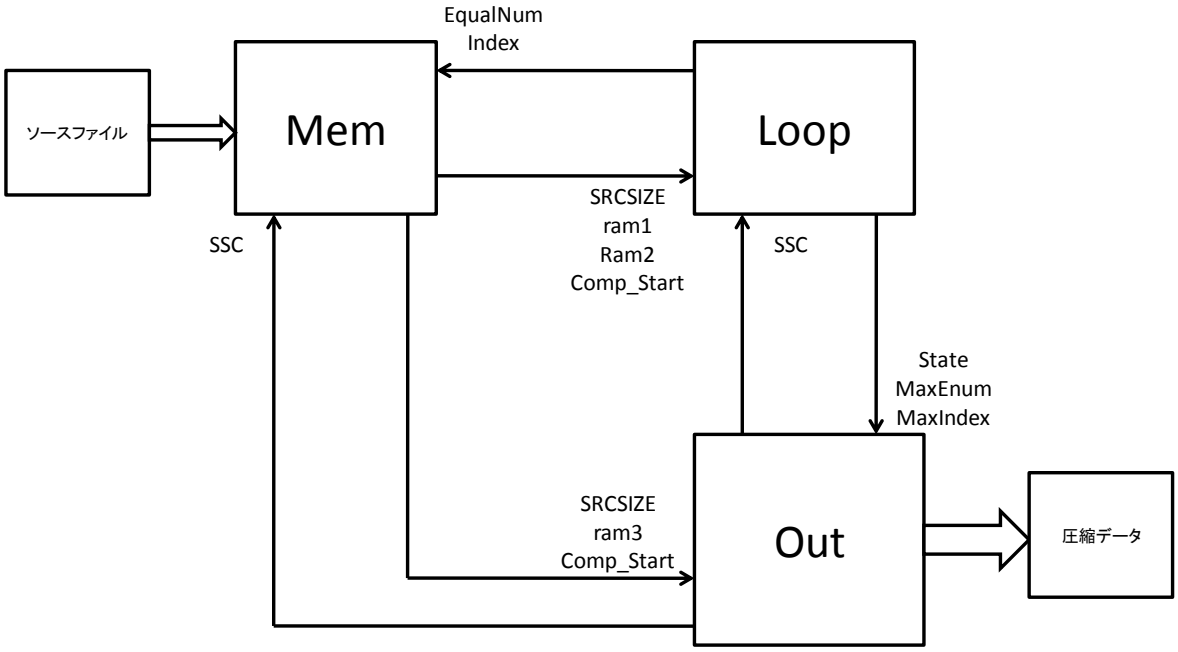


図 3：モジュール分割図

表 3：モジュール間の信号

信号名	詳細
CLK	クロック信号
RST	リセット信号
SSC	参照地点のポインタ
ram1	Memory[Equalnum+SSC]の値
ram2	Memory[Equalnum+SSC-Index]の値
ram3	出力文字 Memory[SSC]
SRCSIZE	テキスト数
Comp_Start	圧縮開始信号
EqualNum	Memory のシークに使用
Index	Memory のシークに使用
MaxEnum	一致文字の最大数
MaxIndex	単語一致部分の最大数
state	モジュールの状態

3.2.1　Top モジュール

Top モジュールの入出力インタフェースを図 4 に、信号線の説明を表 4 に示す。Top モジュールは各モジュールの接続を行うモジュールである。テストベンチから clock 信号と reset 信号を得て、Mem モジュール、Loop モジュール、Out モジュールに信号を伝達する。また、各モジュールを接続するため、多数の wire を持つ。

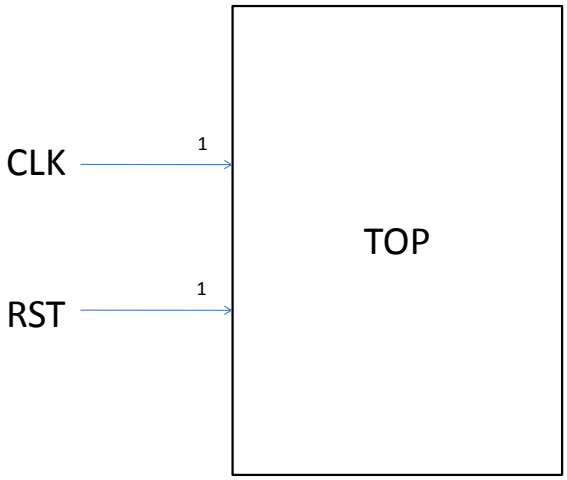


図 4：Top モジュールのインタフェース

表 4：Top モジュールの入出力信号

信号名	方向	幅(bit)	詳細
CLK	Input	1	クロック信号
RST	Input	1	リセット信号

3.2.2　Loop モジュール

Loop モジュールの入出力インタフェースを図 5 に、信号線の説明を表 5 に示す。Loop モジュールでは、Out モジュールからの入力である SSC を参考に、SSC より以前に圧縮可能な文字列が存在しないかどうかの検索（スライド辞書処理）を行う。このブロックは圧縮の主要な処理を行う部分であるため、ハードウェア処理における高速化の効果が最も現れる部分であると考えられる。この Loop モジュールで MaxEnum,MaxIndex を計算し、Out モジュールに信号を出力する。また、圧縮と出力の状態遷移の制御をおこなうのもこのモジュールである。

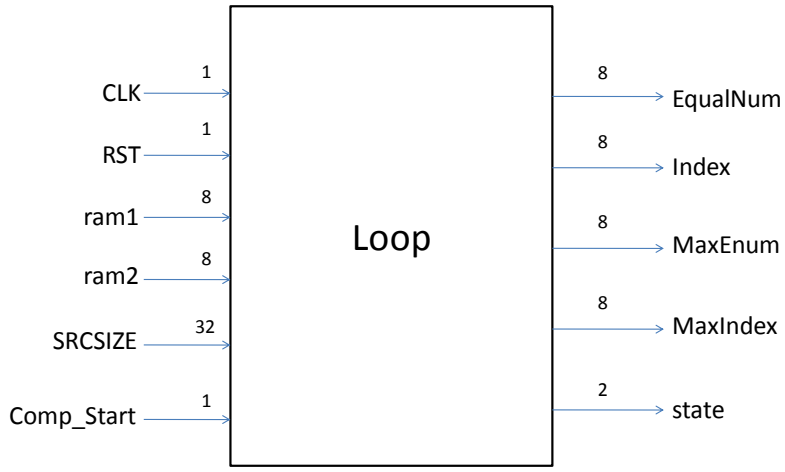


図 5：Loop モジュールのインタフェース

表 5：Loop モジュールの入出力

信号名	方向	幅(bit)	詳細
CLK	Input	1	クロック信号
RST	Input	1	リセット信号
ram1	Input	8	Memory[Equalnum+SSC]の値
ram2	Input	8	Memory[Equalnum+SSC-Index]
SRCSIZE	Input	32	テキスト数
Comp_Start	Input	1	圧縮開始信号
EqualNum	Output	8	Memory のシークに使用
Index	Output	8	Memory のシークに使用
MaxEnum	Output	8	一致文字の最大数
MaxIndex	Output	8	単語一致部分の最大数
state	Output	2	モジュールの状態

3.2.3　Mem モジュール

Mem モジュールの入出力インタフェースを図 6 に、信号線の説明を表 6 に示す。Mem モジュールでは、主にソースファイルからデータを読み込み、配列に ASCII 文字を格納する処理を行う。そのため、回路規模が各モジュール中一番大きくなると考えられる。また、読み込みが完了したことを他のモジュールに comp_start 信号を送信して伝える。これをもとに、他のモジュールは圧縮処理を開始する。その他には、Loop モジュールの条件判定部で使用する ram1,ram2 の値の出力、Out モジュールで使用する ram3 の値を計算し、出力する。

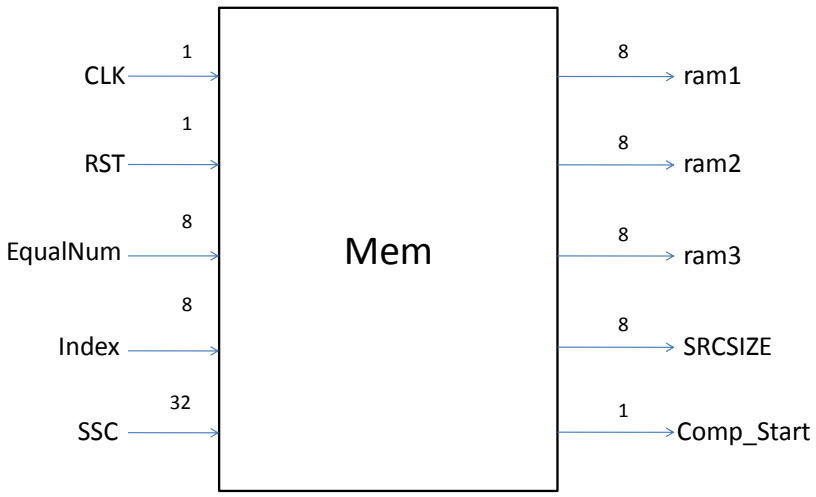


図 6：Mem モジュールのインタフェース

表 6：Mem モジュールの入出力

信号名	方向	幅(bit)	詳細
CLK	Input	1	クロック信号
RST	Input	1	リセット信号
EqualNum	Input	8	Memory[Equalnum+SSC]の値
Index	Input	8	Memory[Equalnum+SSC-Index]
SSC	Input	32	参照地点のポインタ
ram1	Output	8	Memory[Equalnum+SSC]の値
ram2	Output	8	Memory[Equalnum+SSC-Index]
ram3	Output	8	出力文字 Memory[SSC]
SRCSIZE	Output	8	テキスト数
Comp_Start	Input	1	圧縮開始信号

3.2.4　Out モジュール

Out モジュールの入出力インタフェースを図 7 に、信号線の説明を表 7 に示す。Out モジュールは、Loop モジュールで判定した一致文字数により、圧縮情報に変換して出力を行うか、元のデータのまま出力を行うかの判定を行う。また、それらのデータの出力も行うモジュールである。圧縮後の情報をレジスタである ram_o にすべて保持する。また、現在シークしている部分の計算を行い、レジスタ SSC に保存しておく。この値は他のモジュールにも送信され、それぞれの計算に使用される。

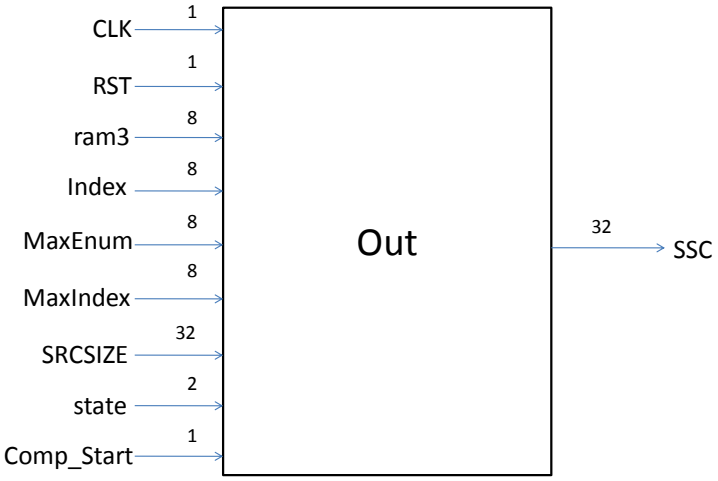


図 7：Out モジュールのインタフェース

表 7：Out モジュールの入出力

信号名	方向	幅(bit)	詳細
CLK	Input	1	クロック信号
RST	Input	1	リセット信号
Ram3	Input	8	出力文字 Memory[SSC]
MaxEnum	Input	8	一致文字の最大数
MaxIndex	Input	8	単語一致部分の最大数
SRCSIZE	Input	32	テキスト数
state	Input	2	モジュールの状態
Comp_Start	Input	1	圧縮開始信号
SSC	Output	32	参照地点のポインタ

3.3　動作検証

3.3.1　処理結果

動作検証用として、以下のような HTML を使用する。全データを掲載することは不可能なので、一部抜粋した。

ソースファイルサイズは 45kB であるが、圧縮を行うことで 27kB まで圧縮することができた。また、デバッグ用に出力した C と Verilog のファイルについても、相違がないことが確認できる。赤で示した部分が圧縮処理された部分である。圧縮部分には①～⑤まで番号を記載した。改行は￥ｎ￥ｒの 2 文字で表す。

・ソースファイル

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                                          ↑① HTML
"http://www.w3.org/TR/html4/loose.dtd">
<html lang="ja">
  ↑②html       ↑③">\n\r<h
<head>
        ↑④ >\n\r<
<meta http>
         ↑⑤http
```

・出力結果

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD □□4.01 Transitional//EN"
                                          ↑①
"http://www.w3.org/TR/html4/loose.dtd">

< □□ lang="ja □□ead        □meta  I□-
  ↑②          ↑③          ↑④     ↑⑤
```

※以下はデバッグ用に ASCII 文字を数値に変えて出力し、確認したもの

・C プログラムでの処理結果

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD[ ][25]+1[6]4.01 Transitional//EN"
                                          ↑①
"http://www.w3.org/TR/html4/loose.dtd">

<[ ][20]+1[4] lang="ja[ ][18]+1[6]ead[ ][8]+1[4]meta [ ][72]+1[4]>
  ↑②                ↑③              ↑④              ↑⑤
```

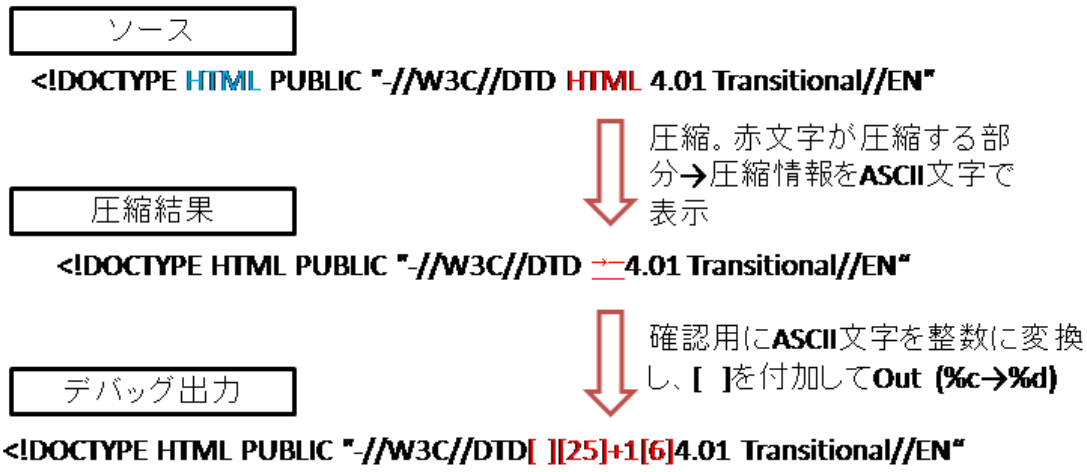
・Verilog での処理結果

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD[ ][ 25]+1[  6]4.01 Transitional//EN"
                                          ↑①
"http://www.w3.org/TR/html4/loose.dtd">

<[ ][ 20]+1[  4] lang="ja[ ][ 18]+1[  6]ead[ ][  8]+1[  4]meta [ ][ 72]+1[  4]>
  ↑②                ↑③              ↑④                ↑⑤
```

3.3.2 検証

実際に 1 行目の①に注目し、検証すると、まずソースの赤文字で示す HTML については 25 文字前に出現しており、空白文字も含め 6 文字が一致している。デバッグ用の結果と照らし合わせると、[ ][25]+1[6]という値となっており、正しい結果が得られている。[ ][25]+1[6]をアスキーで出力したのが、出力結果の文字となっている。



圧縮情報	[ ]	[25]+1	[6]
内容	圧縮情報を示すための文字（NULL文字）	25文字前の単語が一致したという情報 ※圧縮開始情報以上なら+1する	HTMLと前後の空白文字数分のカウント

同様に②html については 20 文字前の 4 文字が一致で　[ ][20]+1[4]

③">\n\r<h は 18 文字前の 6 文字が一致（改行の￥n￥r を含んでいるため）で

[ ][18]+1[6]

④の >\n\r< は 8 文字前の 4 文字が一致で　[ ][8]+1[4]

⑤の http は 72 文字前の 4 文字が一致で[ ][72]+1[4]

いずれも正しい結果が得られている。

4. LZ 圧縮のハード・ソフト分割の検討

4.1 ハードソフト協調設計の手順

ハード・ソフト協調設計は、大きく分けて次のような三つの段階を経て進められる。すなわち、（1）システムの仕様や機能を決定する段階、（2）システムの各機能をハードウェア部とソフトウェア部に分割し資源を割り付ける段階、（3）システムのハードウェア部とソフトウェア部の設計をある程度進めて両部の統合検証をする段階、である。以下にそれぞれを概説する。

（1）では，ハードウェア部とソフトウェア部からなるシステムの機能を記述し、そのシステムが想定通りになるかどうかを検証する。この段階でシステムを記述する際には、あまり詳細には触れない。

例えば「入出力だけ」など、システムの一部の動作のみを記述する。これを「抽象化モデル」と呼ぶ。抽象化モデルの記述には、C 言語/C++、あるいはこれらをベースとしたシステム・レベル記述言語が使われる。

（2）では、各機能をハードウェア部として実現するか、あるいはソフトウェア部として実現するかを決定し(ハード・ソフト分割)し、ソフトウェア部を実現する MPU や DSP を選択する。さらに、ハードウェア部とソフトウェア部（実際には，MPU や DSP）間の通信や、ハードウェア部同士の通信に使うバスの構成もここで決める。

このハード・ソフト分割とバス構成を含む資源の割り付けが、システム全体の性能やコストに大きな影響を与える。ただし、現在の技術レベルでは、各機能をハードウェア部、ソフトウェア部で実行したときの処理性能やチップ・コスト(チップ面積)、消費電力、開発コスト、変更容易性など、多くの要因をこの段階で見積ることが難しいという課題がある。

なお、この段階で MPU や DSP が決まるため，実機に即したソフトウェア開発を始められる。いわゆる、「ハードウェア完成前にソフトウェアの先行開発」が実現する。

（3）では、ハードウェア部、ソフトウェア部の設計を詳細設計の一歩手前まで進める。そして両部の統合検証をクロック・サイクル精度で行う。一般にこの段階では、ハードウェア部は論理合成可能な RTL（register transfer level）記述となる。記述には HDL（hardware description language）を使う。一方ソフトウェア部は、使う予定の MPU や DSP 向けにコンパイルできるような記述となる。

そして、ハードウェア部のデータは論理シミュレータで、ソフトウェア部はMPUやDSPのISS（instruction set simulator）で稼働させる。両シミュレータは協調して動作する。この検証は、ハード・ソフト協調検証（hardware software co-verification）と呼ばれる。[3]

本研究では、分割の流れを図8のようなフローで行った。

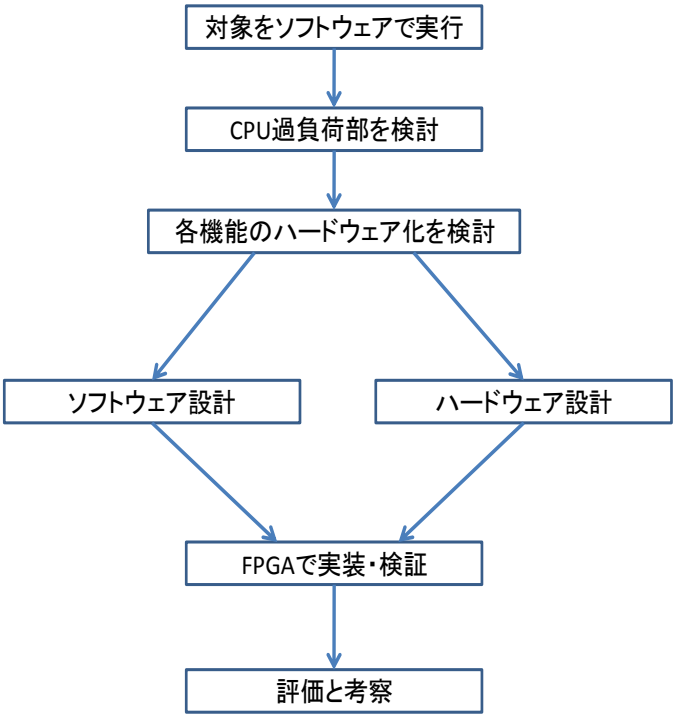


図8：分割フロー

まず対象アプリケーションをC言語で記述してアルゴリズムの理解や正しい結果の確認などを行う。次に、FPGAボード上のMicroBlazeで、ハードウェアモジュール毎に時間計測し、CPU負荷を計測し、負荷の大きい部分を探し出す。その後、各モジュールのハードウェア化について検討してから、性能や回路規模について考慮した上で、分割パターンを選出する。そして実際にソフトウェアとハードウェアの設計を行った上で、FPGAボードに実装し、検証を行う。

4.2　実験と考察

4.2.1　ソフトウェアとハードウェアの性能比較

作成したソフトウェアとハードウェアの処理にかかるクロック数の比較を表 8 に示す。ソフトウェア実行環境はCore2Duo 3.0Ghz、Dram2.0GB、WindowsXP、Cygwin上で

ある。動作時間を計測後、クロック数の算出を行った。ハードウェアについては、ModelSim 上での処理を行った。

表 8：ソフトとハードでの処理時間の比較

モジュール名	Mem	Loop	Out	total
SW クロック数 (M クロック)	93	141	3	235
HW クロック数 (M クロック)	2.02	3.87	0.008	5.9

全体での処理クロック数の比較を行うと、ハードウェア化により、約４０倍の速度向上が得られている。

4.2.2 各ハードウェアの性能評価と考察

作成したハードウェア回路についての評価・比較を行った。表 9 にモジュール単体での回路規模を示す。

表 9：回路規模の比較

モジュール名	Mem	Loop	Out	total
スライス数	303~3479	2147	1482	7108

Mem については、入力するソースに応じて用意するレジスタ数を増やす必要がある。表 9 に示した値は、試験的にレジスタ数を 16 の場合と 2048 で測定した値である。実際に HTML を処理するにあたって、十分なレジスタ数を用意した場合は更に回路規模が増大すると考えられるため、回路規模や汎用性を考えると、この部分はソフトウェア化するべきである。

4.2.3　分割パターンの検討

4.1 節で述べた手法に基づき、分割パターンを検討する。表 10 に分割パターンを決定する上で考慮すべき、各モジュールが回路全体で占める回路規模と SW 実行時の負荷の割合を示す。

表 10：各モジュールが全体に占める回路規模と SW 実行時の負荷割合

	スライス数	SW 実行時の負荷割合
Total	7108	-
Mem	3479	39
Loop	2147	60
Out	1482	1

これらの結果から、表 11 に示すような 4 種類の分割パターンを決定した。

表 11：分割パターン

分割パターン	ハードウェア処理	ソフトウェア処理
A	Loop,Out	Mem
B	Loop	Mem,Out
C	Loop,Mem	Out
D	Mem	Loop,Out

Mem モジュールは回路規模を考慮した場合や、入力ソースが大きく、レジスタ数が膨大になる場合、ソフトウェアで処理することが必然である。これは設計の段階で予想ができるため、実際にハード・ソフト分割を考慮したアプリケーションを作成する場合は、ソフトウェアのみ記述すべきであると言える。Out に関しても、Mem で用意した分のレジスタが必要となるため、回路規模もある程度必要とされる。出力先がソフトであれば、インタフェースを考慮した場合、スムーズなやり取りを行うためにもソフトウェアで実装することが望ましい。ただし、入力データが極端に少ない場合はレジスタ数に応じて回路規模も小さくなるため、設計の単純化を考えると Loop と一つにまとめてハードウェアで処理することも考えられる。Loop に関しては CPU 負荷の割合からハードウェアで処理するのが最適である。

4.2.4　考察

実用的な分割案については、回路規模を考慮した場合 B となる。このパターンでは、Mem と Out をソフトウェアにし、レジスタを完全にソフトウェアに任せるため、どのようなファイルサイズの入力に対しても処理できるという点で優れている。

速度を考慮した場合は分割パターン C が最適であるが、Out の処理時間はソフトウェアでもハードウェアでも非常に少ないため、全てをハードウェアで記述してもほぼ変わらないと言える。

これらの結果から、ハード・ソフトで処理を行う場合はBのパターンで設計することが最適と言え、速度を考慮した場合は完全にハードウェアで処理することが妥当であり、ハード・ソフトで分割するメリットは少ないと言える。ただし、モジュールの分割方法に関してはまだ改善の余地が残っているので、モジュールを更に細かく分割することで、ハード・ソフト分割のメリットが得られる場合があると考えられる。

まとめとして、ハード・ソフト分割で設計する場合は、大量のデータの保持を行う部分はソフトウェアで記述するべきであると言える。今回は、FPGA上で動かすまでには至らなかったが、実機で動作させ、正確な値も測定したい。

5. おわりに

本論文では、LZ圧縮アルゴリズムについて、ハードウェア記述言語Verilog-HDLを用いたアプリケーションの高速化について、またハード/ソフト分割について、分割案の検討を行った。 ハードウェア化による高速化については、LZ法を用いたデータ圧縮プログラムをハードウェア化することで、ソフトウェアの約40倍の速度向上が得られた。

ハード/ソフト最適分割の検討では、LZ圧縮の処理を、４つのモジュールに分割し、それぞれソフトウェアおよびハードウェアモジュールとして実現した。それらを4つの分割パターンを考え、分割案について検討、考察を行い、有用な案を選出した。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導をいただきました山崎勝弘教授に深く感謝いたします。

また、本研究にあたり様々な助言を下さった高性能計算研究室の皆様に心より感謝いたします。

参考文献

[1]土村展之 津留隆之　松井吉光 光成滋生 奥村晴彦 首藤一幸 杉浦方紀　 Java によるアルゴリズム辞典 , 2003

[2]Herbert Schildt　独習 C ,2002

[3]改訂版 EDA 用語辞典 ハードウェア・ソフトウェア協調設計

[4]小林優　入門 Verilog HDL 記述

[5]画像圧縮アルゴリズム LZ 法　http://www2.starcat.ne.jp/~fussy/algo/algo8-5.htm

[6] 梅原直人: Misty1暗号回路の設計とハード/ソフト最適分割の検討
, 立命館大学理工学部情報学科卒業論文, 2007.

[7] 梅原直人: ハード／ソフト最適分割を考慮した AES 暗号システムと JPEG エンコーダの設計と検証, 立命館大学理工学部情報学科卒業論文, 2005.

付録

LZ 圧縮のハードウェア記述

・top モジュール

```
module lz_top(clk,rst);
	input clk;
	input rst;

wire comp_start_w;
wire[1:0] state_w;
wire[7:0]
ram1_w,ram2_w,ram3_w,EqualNum_w,Index_w,MaxEnum_w,MaxIndex_w;
wire[31:0] SRCSIZE_w,SSC_w;


loop loop1(.clk(clk),.rst(rst),.comp_start(comp_start_w),
	.ram1(ram1_w),.ram2(ram2_w),.SRCSIZE(SRCSIZE_w),.SSC(SSC_w),
	.EqualNum_o(EqualNum_w),.Index_o(Index_w),.state_o(state_w),
	.MaxEnum_o(MaxEnum_w),.MaxIndex_o(MaxIndex_w));

out out1(.clk(clk),.rst(rst),.comp_start(comp_start_w),
	.state(state_w),.ram3(ram3_w),.MaxIndex(MaxIndex_w),
	.MaxEnum(MaxEnum_w),.SRCSIZE(SRCSIZE_w),.Index(Index_w),
	.SSC_o(SSC_w));

mem
mem1(.clk(clk),.rst(rst),.comp_start_o(comp_start_w),.EqualNum(EqualNum_w),.
SSC(SSC_w),.Index(Index_w),
	 .ram_o1(ram1_w),.ram_o2(ram2_w),.ram_o3(ram3_w),.SRCSIZE_o(SRCSIZ
E_w));

endmodule
```

・Mem モジュール

```
module lz_top(clk,rst);
	input clk;
	input rst;

wire comp_start_w;
wire[1:0] state_w;
wire[7:0]
ram1_w,ram2_w,ram3_w,EqualNum_w,Index_w,MaxEnum_w,MaxIndex_w;
wire[31:0] SRCSIZE_w,SSC_w;


loop loop1(.clk(clk),.rst(rst),.comp_start(comp_start_w),
	.ram1(ram1_w),.ram2(ram2_w),.SRCSIZE(SRCSIZE_w),.SSC(SSC_w),
	.EqualNum_o(EqualNum_w),.Index_o(Index_w),.state_o(state_w),
	.MaxEnum_o(MaxEnum_w),.MaxIndex_o(MaxIndex_w));
```

```
out out1(.clk(clk),.rst(rst),.comp_start(comp_start_w),
        .state(state_w),.ram3(ram3_w),.MaxIndex(MaxIndex_w),
        .MaxEnum(MaxEnum_w),.SRCSIZE(SRCSIZE_w),.Index(Index_w),
        .SSC_o(SSC_w));

mem
mem1(.clk(clk),.rst(rst),.comp_start_o(comp_start_w),.EqualNum(EqualNum_w),.
SSC(SSC_w),.Index(Index_w),
         .ram_o1(ram1_w),.ram_o2(ram2_w),.ram_o3(ram3_w),.SRCSIZE_o(SRCSIZ
E_w));

endmodule
```

・Loop モジュール

```
module loop(clk,rst,comp_start,ram1,ram2,SRCSIZE,SSC,
        EqualNum_o,Index_o,state_o,MaxEnum_o,MaxIndex_o);
        input clk;
        input rst;
        //mem
        input comp_start;
        input[7:0] ram1,ram2;
        input[31:0] SRCSIZE;
        //out
        input [31:0]SSC;

        //mem
        output[7:0] EqualNum_o,Index_o;
        //out
        output[1:0] state_o;
        output[7:0] MaxEnum_o,MaxIndex_o;


        reg [1:0] state;
        localparam equal = 2'b00;
        localparam equal_break = 2'b01;
        reg [7:0]   MaxEnum,EqualNum;
        reg [7:0]   Index,MaxIndex;

        integer FP3;

initial FP3=$fopen("out.txt");

assign EqualNum_o=EqualNum;
assign Index_o=Index;
assign MaxEnum_o=MaxEnum;
assign MaxIndex_o=MaxIndex;

assign state_o=state;

always@(negedge rst) begin
   if (rst==1'b0)
      begin
```

```
            state<=equal;
            EqualNum<=8'b00000000;
            Index<=8'b00000001;
            MaxEnum<=8'h00;
            MaxIndex<=8'b00000000;
        end
end

always @(posedge clk  ) begin
if(comp_start==1'b1)begin
   /*
    if(SSC>=SRCSIZE)$stop;
    if(writeflag==1)begin
    $fwrite(FP3,"%c",SRCSIZE[7:0]);
    $fwrite(FP3,"%c",SRCSIZE[15:8]);
    $fwrite(FP3,"%c",SRCSIZE[23:16]);
    $fwrite(FP3,"%c",SRCSIZE[31:24]);
    $fwrite(FP3,"%c",PRESSSIZE[7:0]);
    $fwrite(FP3,"%c",PRESSSIZE[15:8]);
    $fwrite(FP3,"%c",PRESSSIZE[23:16]);
    $fwrite(FP3,"%c",PRESSSIZE[31:24]);
    $fwrite(FP3,"%c",ENCODECODE[7:0]);
    $fwrite(FP3,"%c",ENCODECODE[15:8]);
    $fwrite(FP3,"%c",ENCODECODE[23:16]);
    $fwrite(FP3,"%c",ENCODECODE[31:24]);
    writeflag<=1'b0;
   end
   */
if(SSC<SRCSIZE)begin
   //if(Eqflag==1'b1)begin
      case(state)
      equal: begin
        if((EqualNum<Index)
           &&(EqualNum+SSC <= SRCSIZE)
           &&(ram1==ram2)
           )begin
              EqualNum<=EqualNum+8'b00000001;
           end
           else begin
              //Eqflag<=1'b0;
              state<=equal_break;
              Index<=Index+1'b1;
               if((EqualNum>=8'b00000100) && (MaxEnum<EqualNum))begin
                  MaxEnum<=EqualNum;
                  MaxIndex<=Index;
                  end
           end
      end
     //else begin 0116
      equal_break: begin
```

```
            if((Index<=8'd30)
            && (SSC>=Index)
            )begin
            //Eqflag<=1'b1;       0116
            state<=equal;
            EqualNum<=8'b00000000;
            end
            else begin


            Index<=8'b00000001;
            MaxIndex<=8'h00;
            MaxEnum<=8'h00;
            EqualNum<=8'b00000000;
            state<=equal;
          // end 0116
 end
 end
 endcase


end
end
end

endmodule
```

・Out モジュール

```
module out(clk,rst,comp_start,state,ram3,MaxIndex,MaxEnum,SRCSIZE,Index,
        SSC_o);
        input clk;
        input rst;
        input comp_start;
        input[1:0] state;
        input[7:0] ram3,MaxIndex,MaxEnum,Index;
        input[31:0] SRCSIZE;

        output[31:0]SSC_o;

        localparam equal = 2'b00;
        localparam equal_break = 2'b01;

        reg writeflag;
        reg [7:0]  ram_o[0:65535],EncodeCode;
        reg [31:0]  PRESSSIZE,ENCODECODE;
        reg [31:0] SSC,adress_o;

        integer FP3;

initial FP3=$fopen("out.txt");
```

```
assign SSC_o=SSC;

always@(negedge rst) begin
   if (rst==1'b0)
     begin
          SSC<=32'b00000000000000000000000000000000;
          adress_o<=32'b00000000000000000000000000000000;
          EncodeCode<=8'h00;
          PRESSSIZE<=32'b00000000000000000000000000001100;// 25 11001
          ENCODECODE<=32'd00;
          writeflag<=1'b1;
     end
end

always @(posedge clk  ) begin

   if(comp_start==1'b1)begin

      if(SSC>=SRCSIZE)$stop;
      if(writeflag==1)begin
         $fwrite(FP3,"%c",SRCSIZE[7:0]);
         $fwrite(FP3,"%c",SRCSIZE[15:8]);
         $fwrite(FP3,"%c",SRCSIZE[23:16]);
         $fwrite(FP3,"%c",SRCSIZE[31:24]);
         $fwrite(FP3,"%c",PRESSSIZE[7:0]);
         $fwrite(FP3,"%c",PRESSSIZE[15:8]);
         $fwrite(FP3,"%c",PRESSSIZE[23:16]);
         $fwrite(FP3,"%c",PRESSSIZE[31:24]);
         $fwrite(FP3,"%c",ENCODECODE[7:0]);
         $fwrite(FP3,"%c",ENCODECODE[15:8]);
         $fwrite(FP3,"%c",ENCODECODE[23:16]);
         $fwrite(FP3,"%c",ENCODECODE[31:24]);
         writeflag<=1'b0;
         end
      if(SSC<SRCSIZE)begin

         if(!( (Index<=8'd30)
         && (SSC>=Index) )
         )begin

            if(MaxIndex==0)begin
               if(ram3!=EncodeCode)begin
               ram_o[adress_o]<=ram3;
               adress_o<=adress_o+32'd1;
               PRESSSIZE<=PRESSSIZE+32'd1;
               $fwrite(FP3,"%c",ram3);
               end
               else begin
               ram_o[adress_o]<=ram3;
               ram_o[adress_o+32'd1]<=ram3;
               adress_o<=adress_o+32'd1;
```

```
                PRESSSIZE<=PRESSSIZE+32'd2;
                $fwrite(FP3,"%c%c",ram3,ram3);
                end
                SSC<=SSC+32'd1;
              end
              else begin
                ram_o[adress_o]<=EncodeCode;
                if(MaxIndex>=EncodeCode)begin
                  ram_o[adress_o+32'd1]<=(MaxIndex+8'b00000001);
                  ram_o[adress_o+32'd2]<=MaxEnum;

//$fwrite(FP3,"[%c][%c][%c]",EncodeCode,(MaxIndex+8'b00000001),MaxEnum);

$fwrite(FP3,"[%c][%d]+1[%d]",EncodeCode,MaxIndex,MaxEnum);//debug
                end
                else begin
                  ram_o[adress_o+32'd1]<=MaxIndex;
                  ram_o[adress_o+32'd2]<=MaxEnum;
                  $fwrite(FP3,"[%c][%d][%d]",EncodeCode,MaxIndex,MaxEnum);
                end

                PRESSSIZE<=PRESSSIZE+32'd3;
                adress_o<=adress_o+32'd3;
                SSC<=SSC+MaxEnum;

            end

          end
        end
    end //comp start
end//   always

endmodule
```